A Defining SILF in K

SILF is a simple imperative language with functions. It is a C-like language, but a lot simpler in many aspects. For example, SILF has no structures and no pointers. A program consists of a ";"-separated sequence of global variable declarations, followed by a sequence of function declarations, one of them expected to be called main. Nested function declarations are not accepted. The program is executed by calling the function main(). Scoping is static, or lexical, and functions see each other and also the global variables. In particular, a function can recursively call itself. Statements can be grouped in blocks with local variable declarations. The names of the variables that appear in a sequence of declarations are expected to be different, and so are the function names. Additionally, the function names are expected to be different from the global variable names and not to be used for anything else but function invocations. These conditions can be easily checked.

An untyped version of SILF is presented in Appendix A.1, that is, one in which variables and functions are are not declared any types; values in SILF can have two basic types, integer and boolean. Even in its untyped version, SILF is still dynamically typed, in the sense that, when executed, a program "gets stuck" if inapropriate operations are encountered during its execution, such as adding an integer with a boolean, or calling a function with a wrong number of arguments, and so on. By a stuck program we mean one that cannot be advanced anymore due to the fact that no rewrite rule matches. Of course, a variable can be assigned values of different types during its lifetime, and no type checking is performed when functions are invoked, because no typing information is available for them. Note that, for uniformity reasons, we prefer to store the declared functions as lambda-like expressions in the store; that is not necessary (functions could have been stored in a function environment), but it eases the definition of name lookup; indeed, in our curent definitions of SILF discussed in this section, function names and variable names are looked up similarly.

Appendix A.2 adds type declarations to SILF, for both variables and functions. Type declarations are regarded as rigid "contracts", or "specifications", so they cannot be violated and cannot be changed at runtime. The definition in Appendix A.2 modifies the one in Appendix A.1 to check type declarations dynamically. Only the executed path is checked, so there could be other executions possible (under a different input, for example) that violate the typing policy. Values are not tagged with their types in the store; we assume instead a function *typeOf* on integer and boolean values returning their corresponding type. The declared type of a variable is stored in the environment, together with the location of that variable. In the case of arrays, the declared length of the array is incorporated as part of its type, so that runtime array accesses can be checked to fall within the specified boundaries; languages like Java also perform such boundary checks dynamically. Dynamic checking of types, safety policies, contracts, or specifications is common and instructive in its own way. The purpose of Appendix A.2 is to show that runtime verification of safety policies/properties can be very easily achieved in K, by slightly adjusting the existing definition.

Appendix A.3 defines a static type checker for SILF, following the same K framework as the other definitions. The major difference is that the domain of interpretation for the language syntax is now one of types (instead of concrete values), that is, programs or fragments of program, as well as computations in general, now evolve into types; types are therefore the results of computations. Most operations are now strict in all their arguments, including the conditionals and the loops. The program is first traversed and all the global variable and function declarations are collected in the global environment, while the function bodies are scheduled for typing in *toType*. This step is sublinear in complexity, because the bodies of the functions are not traversed. Then each function body is processed separately, traversing all its code; in the worst case, assuming no concurrent capabilities of the underlying rewrite engine, the complexity of checking each function is linear in its size. Complexitywise, our approach to type checking using K is therefore as good as, or better, than defining/implementing type checking algorithms. It has the additional benefit of being formal and using the same logical framework as the definition of the concrete semantics, so it should facilitate formal verification and proofs of correctness (such as, for example, type preservation and progress).

When reading the subsequent sections, recall that the "K-annotated syntax" of a language defines more than the actual syntax of the language: it defines the syntax of that language's computations, including their structural identities (using the "strict" or default attributes; structurally equivalent computations are identical entities), as well as some straghtforward reduction rules (using the "extends" attribute). Even though we use variable names such as d for "declarations" or s for "statements", in K definitions there is

only one syntactic category, K, of computations. In fact, the different equations for sequential composition ";" are given only for clarity; one would suffice. The K annotations to syntax capture local, syntax driven, typically uninteresting and rather trivial and boring parts of a language definition. The remaining rules and equations are grouped into the "K configuration and semantics" part of the definition.

A.1 Untyped SILF

K-Annotated Syntax of Untyped SILF

$$
\begin{array}{rcl}
\mathit{Nat} & ::= & 0 \mid 1 \mid 2 \mid \ldots \text{ all natural numbers} \\
\mathit{Int} & ::= & \ldots \text{ all integer numbers} \\
\mathit{Bool} & ::= & \mathsf{true} \mid \mathsf{false} \\
\mathit{Name} & ::= & \text{all identifiers; to be used as names of variables and functions} \\
\mathit{Exp} & ::= & \mathit{Int} \mid \mathit{Bool} \mid \mathit{Name} \\
 & \mid & \mathit{Name}[\mathit{Exp}]\ [\mathit{strict}(2)] \\
 & \mid & \mathsf{read}() \\
 & \mid & \mathit{Name}(\mathit{List}[\mathit{Exp}])\ [\mathit{strict}(1),\quad f(el_1, e, el_2) \rightleftharpoons e \curvearrowright f(el_1, \Box, el_2)] \\
 & \mid & \mathit{Exp} + \mathit{Exp}\ [\mathit{strict},\ \mathit{extends} +_{\mathit{Int}\times\mathit{Int}\to\mathit{Int}}] \\
 & \mid & \mathit{Exp} - \mathit{Exp}\ [\mathit{strict},\ \mathit{extends} -_{\mathit{Int}\times\mathit{Int}\to\mathit{Int}}] \\
 & \mid & \mathit{Exp} * \mathit{Exp}\ [\mathit{strict},\ \mathit{extends} *_{\mathit{Int}\times\mathit{Int}\to\mathit{Int}}] \\
 & \mid & \mathit{Exp}\,/\,\mathit{Exp}\ [\mathit{strict},\ \mathit{extends}\ \mathit{quotient}_{\mathit{Int}\times\mathit{Int}\to\mathit{Int}}] \\
 & \mid & \mathit{Exp}\,\%\,\mathit{Exp}\ [\mathit{strict},\ \mathit{extends}\ \mathit{remainder}_{\mathit{Int}\times\mathit{Int}\to\mathit{Int}}] \\
 & \mid & -\mathit{Exp}\ [\mathit{strict},\ \mathit{extends} -_{\mathit{Int}\to\mathit{Int}}] \\
 & \mid & \mathit{Exp} < \mathit{Exp}\ [\mathit{strict},\ \mathit{extends}\ <_{\mathit{Int}\times\mathit{Int}\to\mathit{Bool}}] \\
 & \mid & \mathit{Exp} <= \mathit{Exp}\ [\mathit{strict},\ \mathit{extends}\ \leq_{\mathit{Int}\times\mathit{Int}\to\mathit{Bool}}] \\
 & \mid & \mathit{Exp} > \mathit{Exp}\ [\mathit{strict},\ \mathit{extends}\ >_{\mathit{Int}\times\mathit{Int}\to\mathit{Bool}}] \\
 & \mid & \mathit{Exp} >= \mathit{Exp}\ [\mathit{strict},\ \mathit{extends}\ \geq_{\mathit{Int}\times\mathit{Int}\to\mathit{Bool}}] \\
 & \mid & \mathit{Exp} == \mathit{Exp}\ [\mathit{strict},\ \mathit{extends}\ =_{\mathit{Int}\times\mathit{Int}\to\mathit{Bool}}] \\
 & \mid & \mathit{Exp}\ \mathsf{!=}\ \mathit{Exp}\ [\mathit{strict},\ \mathit{extends}\ \neq_{\mathit{Int}\times\mathit{Int}\to\mathit{Bool}}] \\
 & \mid & \mathit{Exp}\ \mathsf{and}\ \mathit{Exp}\ [\mathit{strict},\ \mathit{extends} \wedge_{\mathit{Bool}\times\mathit{Bool}\to\mathit{Bool}}] \\
 & \mid & \mathit{Exp}\ \mathsf{or}\ \mathit{Exp}\ [\mathit{strict},\ \mathit{extends} \vee_{\mathit{Bool}\times\mathit{Bool}\to\mathit{Bool}}] \\
 & \mid & \mathsf{not}\ \mathit{Exp}\ [\mathit{strict},\ \mathit{extends} \neg_{\mathit{Bool}\to\mathit{Bool}}] \\
\mathit{Decl} & ::= & \mathsf{var}\ \mathit{Name} \mid \mathsf{var}\ \mathit{Name}[\mathit{Nat}] \\
 & \mid & \mathit{Decl}; \mathit{Decl}\ [d_1; d_2 = d_1 \curvearrowright d_2] \\
\mathit{Stmt} & ::= & \{\}\ [\{\} = \cdot] \\
 & \mid & \{\mathit{Stmt}\}\ [\{s\} = s] \\
 & \mid & \{\mathit{Decl}; \mathit{Stmt}\} \\
 & \mid & \mathit{Stmt}; \mathit{Stmt}\ [s_1; s_2 = s_1 \curvearrowright s_2] \\
 & \mid & \mathsf{write}(\mathit{Exp})\ [\mathit{strict}] \\
 & \mid & \mathit{Name} = \mathit{Exp}\ [\mathit{strict}(2)] \\
 & \mid & \mathit{Name}[\mathit{Exp}] = \mathit{Exp}\ [\mathit{strict}(2,3)] \\
 & \mid & \mathsf{if}\ \mathit{Exp}\ \mathsf{then}\ \mathit{Stmt}\ \mathsf{else}\ \mathit{Stmt}\ [\mathit{strict}(1)] \\
 & \mid & \mathsf{if}\ \mathit{Exp}\ \mathsf{then}\ \mathit{Stmt}\ [\mathsf{if}\ e\ \mathsf{then}\ s = \mathsf{if}\ e\ \mathsf{then}\ s\ \mathsf{else}\ \cdot] \\
 & \mid & \mathsf{while}\ \mathit{Exp}\ \mathsf{do}\ \mathit{Stmt} \\
 & \mid & \mathsf{for}\ \mathit{Name} = \mathit{Exp}\ \mathsf{to}\ \mathit{Exp}\ \mathsf{do}\ \mathit{Stmt} \\
 & & \quad [\mathsf{for}\ i = e_1\ \mathsf{to}\ e_2\ \mathsf{do}\ s = \{\mathsf{var}\ i;\ i = e_1;\ \mathsf{while}\ (i <= e_2)\ \mathsf{do}\ \{s; i = i + 1\}\}] \\
 & \mid & \mathsf{call}\ \mathit{Exp}\ [\mathit{strict}] \\
 & \mid & \mathsf{return}\ \mathit{Exp}\ [\mathit{strict}] \\
\mathit{FunDecl} & ::= & \mathsf{function}\ \mathit{Name}(\mathit{List}[\mathit{Name}])\ \mathit{Stmt} \\
 & \mid & \mathit{FunDecl}\,\mathit{FunDecl}\ [fd_1\ fd_2 = fd_1 \curvearrowright fd_2] \\
\mathit{Pgm} & ::= & \mathit{FunDecl} \\
 & \mid & \mathit{Decl}; \mathit{FunDecl}\ [d; fd = d \curvearrowright fd]
\end{array}
$$

K Configuration and Semantics of Untyped SILF

$$
\begin{array}{rcl}
\mathit{Val} & ::= & \mathit{Int} \mid \mathit{Bool} \mid \lambda \mathit{List}[\mathit{Name}].K \\
\mathit{KResult} & ::= & \mathit{Val} \\
\mathit{Env} & ::= & \mathit{Map}[\mathit{Name}, \mathit{Loc}] \\
\mathit{Store} & ::= & \mathit{Map}[\mathit{Loc}, \mathit{Val}] \\
\mathit{FStack} & ::= & \mathit{List}[\mathit{Env} \times K] \\
\mathit{ConfigItem} & ::= & k(K) \mid \mathit{fstack}(\mathit{FStack}) \mid \mathit{env}(\mathit{Env}) \mid \mathit{genv}(\mathit{Env}) \\
& \mid & \mathit{in}(\mathit{List}[\mathit{Int}]) \mid \mathit{out}(\mathit{List}[\mathit{Int}]) \mid \mathit{store}(\mathit{Store}) \mid \mathit{nextLoc}(\mathit{Loc}) \\
\mathit{Config} & ::= & \mathit{List}[\mathit{Int}] \mid [\![K, \mathit{List}[\mathit{Int}]]\!] \mid [\![\mathit{Set}[\mathit{ConfigItem}]]\!] \\
& \mid & [\![K]\!] \; [[\![k]\!] = [\![k, \cdot]\!]] \\
K & ::= & \ldots \mid \mathit{run} \mid \mathit{restore}(\mathit{Env})
\end{array}
$$

$[\![p, il]\!] = [\![k(p \curvearrowright \mathit{run}) \; \mathit{fstack}(\cdot) \; \mathit{env}(\cdot) \; \mathit{genv}(\cdot) \; \mathit{in}(il) \; \mathit{out}(\cdot) \; \mathit{store}(\cdot) \; \mathit{nextLoc}(\mathit{loc}(0))]\!]$
$[\![\langle k(\cdot) \; \mathit{out}(il)\rangle]\!] = il$
$k(\underbrace{\mathit{run}}_{\mathsf{call\ main}()}) \; \mathit{env}(\rho) \; \mathit{genv}(\underbrace{\cdot}_{\rho})$

$k(\underbrace{x\rangle}_{\sigma[\rho[x]]} \; \mathit{env}(\rho) \; \mathit{store}(\sigma)$
$k(\underbrace{x[n]\rangle}_{\sigma[\rho[x] +_{Loc} n]} \; \mathit{env}(\rho) \; \mathit{store}(\sigma)$
$k(\underbrace{\mathsf{read}()\rangle}_{i} \; \mathit{in}(\underbrace{i\rangle}_{\cdot})$
$k(\underbrace{(\lambda xl.s)(vl) \curvearrowright k)}_{s} \; \mathit{fstack}(\underbrace{\cdot\rangle}_{(\rho,k)}) \; \mathit{env}(\underbrace{\rho}_{\rho'[xl \leftarrow ll']}) \; \mathit{genv}(\rho') \; \mathit{store}(\underbrace{\sigma}_{\sigma[ll' \leftarrow vl]}) \; \mathit{nextLoc}(\underbrace{l}_{l'})$ where l' is $l +_{Loc} |xl|$, and ll' is $l \ldots (l' +_{Loc} -1)$
$k(\underbrace{\mathsf{var}\ x\rangle}_{\cdot} \; \mathit{env}(\underbrace{\rho}_{\rho[x \leftarrow l]}) \; \mathit{nextLoc}(\underbrace{l}_{l +_{Loc} 1})$
$k(\underbrace{\mathsf{var}\ x[n]\rangle}_{\cdot} \; \mathit{env}(\underbrace{\rho}_{\rho[x \leftarrow l]}) \; \mathit{nextLoc}(\underbrace{l}_{l +_{Loc} n})$
$k(\underbrace{\{d; s\}}_{d \curvearrowright s \curvearrowright \mathit{restore}(\rho)}\rangle \; \mathit{env}(\rho)$
$k(\underbrace{\mathit{restore}(\rho)\rangle}_{\cdot} \; \mathit{env}(\underbrace{\_}_{\rho})$
$k(\underbrace{\mathsf{write}\ i\rangle}_{\cdot} \; \mathit{out}(\underbrace{\cdot}_{i})$
$k(\underbrace{x = v\rangle}_{\cdot} \; \mathit{env}(\rho) \; \mathit{store}(\underbrace{\sigma}_{\sigma[\rho[x] \leftarrow v]})$
$k(\underbrace{x[n] = v\rangle}_{\cdot} \; \mathit{env}(\rho) \; \mathit{store}(\underbrace{\sigma}_{\sigma[\rho[x] +_{Loc} n \leftarrow v]})$
$\mathsf{if\ true\ then}\ s_1\ \mathsf{else}\ s_2 \rightarrow s_1$
$\mathsf{if\ false\ then}\ s_1\ \mathsf{else}\ s_2 \rightarrow s_2$
$k(\mathsf{while}\ b\ \mathsf{do}\ s\rangle = k(\mathsf{if}\ b\ \mathsf{then}\ (s; \mathsf{while}\ b\ \mathsf{do}\ s)\rangle$
$\mathsf{call}\ v \rightarrow .$
$k(\underbrace{\mathsf{return}\ v \curvearrowright \_}_{v \curvearrowright k}) \; \mathit{fstack}(\underbrace{(\rho, k)\rangle}_{\cdot}) \; \mathit{env}(\underbrace{\_}_{\rho})$
$k(\underbrace{\mathsf{function}\ f(xl)\ s\rangle}_{\cdot} \; \mathit{env}(\underbrace{\rho}_{\rho[f \leftarrow l]}) \; \mathit{store}(\underbrace{\sigma}_{\sigma[l \leftarrow \lambda xl.s]}) \; \mathit{nextLoc}(\underbrace{l}_{l +_{Loc} 1})$

A.2 Type Checking SILF Dynamically

K-Annotated Syntax of Dynamically Type-Checked SILF

$$
\begin{array}{rcl}
\mathit{Nat} & ::= & 0 \mid 1 \mid 2 \mid \ldots \text{ all natural numbers} \\
\mathit{Int} & ::= & \ldots \text{ all integer numbers} \\
\mathit{Bool} & ::= & \mathsf{true} \mid \mathsf{false} \\
\mathit{Name} & ::= & \text{all identifiers; to be used as names of variables and functions} \\
\mathit{Type} & ::= & \mathit{int} \mid \mathit{bool} \text{ (one may add more types if one extends the language)} \\
\mathit{Exp} & ::= & \mathit{Int} \mid \mathit{Bool} \\
 & \mid & \mathit{Name} \\
 & \mid & \mathit{Name}[\mathit{Exp}]\ [\mathit{strict}(2)] \\
 & \mid & \mathsf{read}() \\
 & \mid & \mathit{Name}(\mathit{List}[\mathit{Exp}])\ [\mathit{strict}(1),\ \ f(el_1, e, el_2) \rightharpoonup e \curvearrowright f(el_1, \Box, el_2)] \\
 & \mid & \mathit{Exp} + \mathit{Exp}\ [\mathit{strict},\ \mathit{extends}\ +_{\mathit{Int}\times\mathit{Int}\to\mathit{Int}}] \\
 & \mid & \mathit{Exp} - \mathit{Exp}\ [\mathit{strict},\ \mathit{extends}\ -_{\mathit{Int}\times\mathit{Int}\to\mathit{Int}}] \\
 & \mid & \mathit{Exp} * \mathit{Exp}\ [\mathit{strict},\ \mathit{extends}\ *_{\mathit{Int}\times\mathit{Int}\to\mathit{Int}}] \\
 & \mid & \mathit{Exp}\,/\,\mathit{Exp}\ [\mathit{strict},\ \mathit{extends}\ \mathit{quotient}_{\mathit{Int}\times\mathit{Int}\to\mathit{Int}}] \\
 & \mid & \mathit{Exp}\,\%\,\mathit{Exp}\ [\mathit{strict},\ \mathit{extends}\ \mathit{remainder}_{\mathit{Int}\times\mathit{Int}\to\mathit{Int}}] \\
 & \mid & -\mathit{Exp}\ [\mathit{strict},\ \mathit{extends}\ -_{\mathit{Int}\to\mathit{Int}}] \\
 & \mid & \mathit{Exp} < \mathit{Exp}\ [\mathit{strict},\ \mathit{extends}\ <_{\mathit{Int}\times\mathit{Int}\to\mathit{Bool}}] \\
 & \mid & \mathit{Exp} <= \mathit{Exp}\ [\mathit{strict},\ \mathit{extends}\ \leq_{\mathit{Int}\times\mathit{Int}\to\mathit{Bool}}] \\
 & \mid & \mathit{Exp} > \mathit{Exp}\ [\mathit{strict},\ \mathit{extends}\ >_{\mathit{Int}\times\mathit{Int}\to\mathit{Bool}}] \\
 & \mid & \mathit{Exp} >= \mathit{Exp}\ [\mathit{strict},\ \mathit{extends}\ \geq_{\mathit{Int}\times\mathit{Int}\to\mathit{Bool}}] \\
 & \mid & \mathit{Exp} == \mathit{Exp}\ [\mathit{strict},\ \mathit{extends}\ =_{\mathit{Int}\times\mathit{Int}\to\mathit{Bool}}] \\
 & \mid & \mathit{Exp}\ \mathsf{!=}\ \mathit{Exp}\ [\mathit{strict},\ \mathit{extends}\ \neq_{\mathit{Int}\times\mathit{Int}\to\mathit{Bool}}] \\
 & \mid & \mathit{Exp}\ \mathsf{and}\ \mathit{Exp}\ [\mathit{strict},\ \mathit{extends}\ \wedge_{\mathit{Bool}\times\mathit{Bool}\to\mathit{Bool}}] \\
 & \mid & \mathit{Exp}\ \mathsf{or}\ \mathit{Exp}\ [\mathit{strict},\ \mathit{extends}\ \vee_{\mathit{Bool}\times\mathit{Bool}\to\mathit{Bool}}] \\
 & \mid & \mathsf{not}\ \mathit{Exp}\ [\mathit{strict},\ \mathit{extends}\ \neg_{\mathit{Bool}\to\mathit{Bool}}] \\
\mathit{Decl} & ::= & \mathsf{var}\ \mathit{Type}\ \mathit{Name} \mid \mathsf{var}\ \mathit{Type}\ \mathit{Name}[\mathit{Nat}] \\
 & \mid & \mathit{Decl}; \mathit{Decl}\ [d_1; d_2 = d_1 \curvearrowright d_2] \\
\mathit{Stmt} & ::= & \{\}\ [\{\} = \cdot] \\
 & \mid & \{\mathit{Stmt}\}\ [\{s\} = s] \\
 & \mid & \{\mathit{Decl}; \mathit{Stmt}\} \\
 & \mid & \mathit{Stmt}; \mathit{Stmt}\ [s_1; s_2 = s_1 \curvearrowright s_2] \\
 & \mid & \mathsf{write}(\mathit{Exp})\ [\mathit{strict}] \\
 & \mid & \mathit{Name} = \mathit{Exp}\ [\mathit{strict}(2)] \\
 & \mid & \mathit{Name}[\mathit{Exp}] = \mathit{Exp}\ [\mathit{strict}(2,3)] \\
 & \mid & \mathsf{if}\ \mathit{Exp}\ \mathsf{then}\ \mathit{Stmt}\ \mathsf{else}\ \mathit{Stmt}\ [\mathit{strict}(1)] \\
 & \mid & \mathsf{if}\ \mathit{Exp}\ \mathsf{then}\ \mathit{Stmt}\ [\mathsf{if}\, e\ \mathsf{then}\, s = \mathsf{if}\, e\ \mathsf{then}\, s\ \mathsf{else}\, \cdot] \\
 & \mid & \mathsf{while}\ \mathit{Exp}\ \mathsf{do}\ \mathit{Stmt} \\
 & \mid & \mathsf{for}\ \mathit{Name} = \mathit{Exp}\ \mathsf{to}\ \mathit{Exp}\ \mathsf{do}\ \mathit{Stmt} \\
 & & \quad [\mathsf{for}\ i = e_1\ \mathsf{to}\ e_2\ \mathsf{do}\ s = \{\mathsf{var}\ \mathit{int}\ i;\ i = e_1;\ \mathsf{while}\ (i <= e_2)\ \mathsf{do}\ \{s; i = i + 1\}\}] \\
 & \mid & \mathsf{call}\ \mathit{Exp}\ [\mathit{strict}] \\
 & \mid & \mathsf{return}\ \mathit{Exp}\ [\mathit{strict}] \\
\mathit{FunDecl} & ::= & \mathsf{function}\ \mathit{Type}\ \mathit{Name}(\mathit{List}[\mathit{Type}]\ \mathit{List}[\mathit{Name}])\ \mathit{Stmt} \\
 & \mid & \mathit{FunDecl}\,\mathit{FunDecl}\ [\mathit{fd}_1\ \mathit{fd}_2 = \mathit{fd}_1 \curvearrowright \mathit{fd}_2] \\
\mathit{Pgm} & ::= & \mathit{FunDecl} \\
 & \mid & \mathit{Decl}; \mathit{FunDecl}\ [d; \mathit{fd} = d \curvearrowright \mathit{fd}]
\end{array}
$$

K Configuration and Semantics of Dynamically Type-Checked SILF

$$
\begin{array}{rcl}
\mathit{Val} & ::= & \mathit{Int} \mid \mathit{Bool} \mid \mathit{Type}\ \lambda \mathit{List}[\mathit{Type}]\ \mathit{List}[\mathit{Name}].K \\
\mathit{KResult} & ::= & \mathit{Val} \\
\mathit{Env} & ::= & \mathit{Map}[\mathit{Name}, \mathit{Loc} \times \mathit{Type}] \quad (\text{let } \rho[x] \text{ be } (\mathit{loc}(\rho[x]), \mathit{type}(\rho[x])) \text{ for each } \rho \in \mathit{Env}, x \in \mathit{Name}) \\
\mathit{Store} & ::= & \mathit{Map}[\mathit{Loc}, \mathit{Val}] \\
\mathit{FStack} & ::= & \mathit{List}[\mathit{Env} \times K \times \mathit{Type}] \\
\mathit{ConfigItem} & ::= & k(K) \mid \mathit{fstack}(\mathit{FStack}) \mid \mathit{env}(\mathit{Env}) \mid \mathit{genv}(\mathit{Env}) \\
& \mid & \mathit{in}(\mathit{List}[\mathit{Int}]) \mid \mathit{out}(\mathit{List}[\mathit{Int}]) \mid \mathit{return}(\mathit{Type}) \mid \mathit{store}(\mathit{Store}) \mid \mathit{nextLoc}(\mathit{Loc}) \\
\mathit{Config} & ::= & \mathit{List}[\mathit{Int}] \mid [\![K, \mathit{List}[\mathit{Int}]]\!] \mid [\![\mathit{Set}[\mathit{ConfigItem}]]\!] \\
& \mid & [\![K]\!]\ [[\![k]\!] = [\![k, \cdot]\!]] \\
K & ::= & \ldots \mid \mathit{run} \mid \mathit{restore}(\mathit{Env}) \\
\mathit{Type} & ::= & \ldots \mid ? \mid \mathit{Type}[\mathit{Nat}]
\end{array}
$$

$[\![p, il]\!] = [\![k(p \curvearrowright \mathit{run})\ \mathit{fstack}(\cdot)\ \mathit{env}(\cdot)\ \mathit{genv}(\cdot)\ \mathit{in}(il)\ \mathit{out}(\cdot)\ \mathit{return}(?)\ \mathit{store}(\cdot)\ \mathit{nextLoc}(\mathit{loc}(0))]\!]$

$[\![\langle k(\cdot)\ \mathit{out}(il)\rangle]\!] = il$

$k(\underbrace{\mathit{run}}_{\mathsf{call\ main}()})\ \mathit{env}(\rho)\ \mathit{genv}(\underbrace{\cdot}_{\rho})$

$k(\underbrace{x}_{\sigma[\mathit{loc}(\rho[x])]}\rangle\ \mathit{env}(\rho)\ \mathit{store}(\sigma)$

$k(\underbrace{x[n]}_{\sigma[\mathit{loc}(\rho[x]) +_{\mathit{Loc}} n]}\rangle\ \mathit{env}(\rho)\ \mathit{store}(\sigma)$ where $\mathit{type}(\rho[x]) = t[n']$ with $n < n'$

$k(\underbrace{\mathsf{read}()}_{i}\rangle\ \mathit{in}(\underbrace{i}_{\cdot}\rangle$

$k(\underbrace{(t\ \lambda tl\, xl.s)(vl) \curvearrowright k}_{s})\ \mathit{fstack}(\underbrace{\cdot}_{(\rho, k, t')}\rangle\ \mathit{env}(\underbrace{\rho}_{\rho'[xl \leftarrow (ll', tl)]})\ \mathit{genv}(\rho')\ \mathit{return}(\underbrace{t'}_{t})\ \mathit{store}(\underbrace{\sigma}_{\sigma[ll' \leftarrow vl]})\ \mathit{nextLoc}(\underbrace{l}_{l'})$

where l' is $l +_{\mathit{Loc}} |xl|$, ll' is $l \ldots (l +_{\mathit{Loc}} -1)$, and $\mathit{typeOf}(vl){=}tl$

$k(\underbrace{\mathsf{var}\ t\, x}_{\cdot}\rangle\ \mathit{env}(\underbrace{\rho}_{\rho[x \leftarrow (l, t)]})\ \mathit{nextLoc}(\underbrace{l}_{l +_{\mathit{Loc}} 1})$

$k(\underbrace{\mathsf{var}\ t\, x[n]}_{\cdot}\rangle\ \mathit{env}(\underbrace{\rho}_{\rho[x \leftarrow (l, t[n])]})\ \mathit{nextLoc}(\underbrace{l}_{l +_{\mathit{Loc}} n})$

$k(\underbrace{\{d; s\}}_{d \curvearrowright s \curvearrowright \mathit{restore}(\rho)}\rangle\ \mathit{env}(\rho)$

$k(\underbrace{\mathit{restore}(\rho)}_{\cdot}\rangle\ \mathit{env}(\underbrace{\_}_{\rho})$

$k(\underbrace{\mathsf{write}\ i}_{\cdot}\rangle\ \mathit{out}(\underbrace{\cdot}_{i})$

$k(\underbrace{x = v}_{\cdot}\rangle\ \mathit{env}(\rho)\ \mathit{store}(\underbrace{\sigma}_{\sigma[\mathit{loc}(\rho[x]) \leftarrow v]})$ where $\mathit{type}(\rho[x]) = \mathit{typeOf}(v)$

$k(\underbrace{x[n] = v}_{\cdot}\rangle\ \mathit{env}(\rho)\ \mathit{store}(\underbrace{\sigma}_{\sigma[\mathit{loc}(\rho[x]) +_{\mathit{Loc}} n \leftarrow v]})$ where $\mathit{type}(\rho[x]) = \mathit{typeOf}(v)[n']$ with $n < n'$

$\mathsf{if\ true\ then}\ s_1\ \mathsf{else}\ s_2 \rightarrow s_1$

$\mathsf{if\ false\ then}\ s_1\ \mathsf{else}\ s_2 \rightarrow s_2$

$k(\mathsf{while}\ b\ \mathsf{do}\ s\rangle = k(\mathsf{if}\ b\ \mathsf{then}\ (s; \mathsf{while}\ b\ \mathsf{do}\ s)\rangle$

$\mathsf{call}\ v \rightarrow .$

$k(\underbrace{\mathsf{return}\ v \curvearrowright \_}_{v \curvearrowright k})\ \mathit{fstack}(\underbrace{(\rho, k, t)\rangle}_{\cdot})\ \mathit{env}(\underbrace{\_}_{\rho})\ \mathit{return}(\underbrace{t'}_{t})$ if $\mathit{typeOf}(v) = t'$

$k(\underbrace{\mathsf{function}\ t\ f(tl\ xl)\ s}_{\cdot}\rangle\ \mathit{env}(\underbrace{\rho}_{\rho[f \leftarrow (l, ?)]})\ \mathit{store}(\underbrace{\sigma}_{\sigma[l \leftarrow t\ \lambda tl\ xl.s]})\ \mathit{nextLoc}(\underbrace{l}_{l +_{\mathit{Loc}} 1})$

A.3 Type Checking SILF Statically

K-Annotated Syntax of a Static Type Checker for SILF

Nat	::=	$0 \mid 1 \mid 2 \mid \ldots$ all natural numbers
Int	::=	$\ldots$ all integer numbers
Bool	::=	true | false
Name	::=	all identifiers; to be used as names of variables and functions
Type	::=	*int* | *bool* (one may add more types if one extends the language)
Exp	::=	*Int* | *Bool*
	|	*Name*
	|	*Name*[*Exp*] [*strict*]
	|	read()
	|	*Name*(*List*[*Exp*]) [*strict*(1), $f(el_1, e, el_2) \rightleftharpoons e \curvearrowright f(el_1, \square, el_2)$]
	|	*Exp* + *Exp* [*strict*]
	|	*Exp* − *Exp* [*strict*]
	|	*Exp* ∗ *Exp* [*strict*]
	|	*Exp* / *Exp* [*strict*]
	|	*Exp*%*Exp* [*strict*]
	|	−*Exp* [*strict*]
	|	*Exp* < *Exp* [*strict*]
	|	*Exp* <= *Exp* [*strict*]
	|	*Exp* > *Exp* [*strict*]
	|	*Exp* >= *Exp* [*strict*]
	|	*Exp* == *Exp* [*strict*]
	|	*Exp* != *Exp* [*strict*]
	|	*Exp* and *Exp* [*strict*]
	|	*Exp* or *Exp* [*strict*]
	|	not *Exp* [*strict*]
Decl	::=	var *Type Name* | var *Type Name*[*Nat*]
	|	*Decl*; *Decl* [*strict*]
Stmt	::=	{} (typing of blocks counts as reduction steps, so they are defined in the semantics part)
	|	{*Stmt*} [*strict*]
	|	{*Decl*; *Stmt*}
	|	*Stmt*; *Stmt* [*strict*]
	|	write(*Exp*) [*strict*]
	|	*Name* = *Exp* [*strict*]
	|	*Name*[*Exp*] = *Exp* [*strict*]
	|	if *Exp* then *Stmt* else *Stmt* [*strict*]
	|	if *Exp* then *Stmt* [*strict*]
	|	while *Exp* do *Stmt* [*strict*]
	|	for *Name* = *Exp* to *Exp* do *Stmt* [for $i = e_1$ to e_2 do $s = \{$var *int* i; $i = e_1$; while $(i <= e_2)$ do $\{s; i = i + 1\}\}$]
	|	call *Exp* [*strict*]
	|	return *Exp* [*strict*]
FunDecl	::=	function *Type Name*(*List*[*Type*] *List*[*Name*]) *Stmt*
	|	*FunDecl FunDecl* [*strict*]
Pgm	::=	*FunDecl*
	|	*Decl*; *FunDecl* [*strict*]

K Configuration and Semantics of a Static Type Checker for SILF

$$
\begin{array}{rcl}
\mathit{Type} & ::= & \ldots \mid ? \mid \mathit{decl} \mid \mathit{stmt} \mid \mathit{fundecl} \mid \mathit{pgm} \mid \mathit{Type}[] \mid \mathit{List}[\mathit{Type}] \to \mathit{Type} \\
\mathit{KResult} & ::= & \mathit{Type} \\
\mathit{TEnv} & ::= & \mathit{Map}[\mathit{Name}, \mathit{Type}] \\
\mathit{ConfigItem} & ::= & k(K) \mid \mathit{tenv}(\mathit{TEnv}) \mid \mathit{gtenv}(\mathit{TEnv}) \mid \mathit{return}(\mathit{Type}) \mid \mathit{toType}(\cdot) \\
\mathit{Config} & ::= & \mathit{done} \mid [\![K]\!] \mid [\![\mathit{Set}[\mathit{ConfigItem}]]\!] \\
K & ::= & \ldots \mid \mathit{restore}(\mathit{TEnv}) \mid \mathit{Type}\ \lambda\mathit{List}[\mathit{Type}]\ \mathit{List}[\mathit{Name}].K
\end{array}
$$

$[\![p]\!] = [\![k(p)\ \mathit{tenv}(\cdot)\ \mathit{gtenv}(\cdot)\ \mathit{return}(?)\ \mathit{toType}(\cdot)]\!]$
$[\![\langle k(\cdot)\ \mathit{toType}(\cdot)\rangle]\!] = \mathit{done}$

$i \to \mathit{int},\ b \to \mathit{bool}$
$k(\underline{\ x\ }\rangle\ \mathit{tenv}(\rho)$
$\quad\overline{\rho[x]}$
$(t[])[\mathit{int}] \to t$
$\mathsf{read}() \to \mathit{int}$
$(\mathit{tl} \to t)(\mathit{tl}) \to t$
$\mathit{int} + \mathit{int} \to \mathit{int},\ \mathit{int} - \mathit{int} \to \mathit{int},\ \mathit{int} * \mathit{int} \to \mathit{int},\ \mathit{int} / \mathit{int} \to \mathit{int},\ \mathit{int}\ \%\ \mathit{int} \to \mathit{int},\ -\mathit{int} \to \mathit{int}$
$\mathit{int} < \mathit{int} \to \mathit{bool},\ \mathit{int} <= \mathit{int} \to \mathit{bool},\ \mathit{int} > \mathit{int} \to \mathit{bool},\ \mathit{int} >= \mathit{int} \to \mathit{bool},\ \mathit{int} == \mathit{int} \to \mathit{bool},\ \mathit{int}\ \mathsf{!=}\ \mathit{int} \to \mathit{bool}$
$\mathit{bool}\ \mathsf{and}\ \mathit{bool} \to \mathit{bool},\ \mathit{bool}\ \mathsf{or}\ \mathit{bool} \to \mathit{bool},\ \mathsf{not}\ \mathit{bool} \to \mathit{bool}$
$k(\underline{\mathsf{var}\ t\ x}\rangle\ \mathit{tenv}(\underline{\quad\rho\quad})$
$\quad\overline{\mathit{decl}}\qquad\overline{\rho[x \leftarrow t]}$
$k(\underline{\mathsf{var}\ t\ x[\mathit{int}]}\rangle\ \mathit{tenv}(\underline{\quad\rho\quad})$
$\quad\overline{\mathit{decl}}\qquad\overline{\rho[x \leftarrow t[]]}$
$\mathit{decl}; \mathit{decl} \to \mathit{decl}$
$\{\} \to \mathit{stmt}$
$\{\mathit{stmt}\} \to \mathit{stmt}$
$k(\underline{\quad\{d; s\}\quad}\rangle\ \mathit{tenv}(\rho)$ (this and the two above are rules now, because they count as reduction steps)
$\quad\overline{d; s \curvearrowright \mathit{restore}(\rho)}$
$\mathit{decl}; \mathit{stmt} \to \mathit{stmt}$
$k(t \curvearrowright \underline{\mathit{restore}(\rho)}\rangle\ \mathit{tenv}(\underline{\_})$
$\qquad\cdot\qquad\qquad\overline{\rho}$
$\mathsf{write}\ \mathit{int} \to \mathit{int}$
$(t = t) \to \mathit{stmt}$
$((t[])[\mathit{int}] = t) \to \mathit{stmt}$
$\mathsf{if}\ \mathit{bool}\ \mathsf{then}\ \mathit{stmt}\ \mathsf{else}\ \mathit{stmt} \to \mathit{stmt}$
$\mathsf{if}\ \mathit{bool}\ \mathsf{then}\ \mathit{stmt} \to \mathit{stmt}$
$\mathsf{while}\ \mathit{bool}\ \mathsf{do}\ \mathit{stmt} \to \mathit{stmt}$
$\mathsf{call}\ t \to \mathit{stmt}$
$k(\underline{\mathsf{return}\ t \curvearrowright \_}) \ \mathit{return}(\underline{t})$
$\qquad\cdot\qquad\qquad\overline{?}$
$k(\underline{\mathsf{function}\ t\ f(\mathit{tl}\ \mathit{xl})\ s}\rangle\ \mathit{tenv}(\underline{\quad\rho\quad})\ \mathit{toType}\langle\underline{\quad\cdot\quad})$
$\qquad\overline{\mathit{fundecl}}\qquad\qquad\overline{\rho[f \leftarrow (\mathit{tl} \to t)]}\qquad\qquad\overline{t\ \lambda\mathit{tl}\,\mathit{xl}.s}$
$\mathit{fundecl}\ \mathit{fundecl} \to \mathit{fundecl}$
$\mathit{decl}; \mathit{fundecl} \to \mathit{pgm}$
$k(\underline{t})\ \mathit{tenv}(\rho)\ \mathit{gtenv}(\underline{\cdot})$ if t is pgm or $\mathit{fundecl}$
$\quad\cdot\qquad\qquad\overline{\rho}$
$k(\underline{\cdot})\ \mathit{tenv}(\underline{\quad\rho\quad})\ \mathit{gtenv}(\rho')\ \mathit{return}(\underline{t'})\ \mathit{toType}(\underline{t\ \lambda\mathit{tl}\ \mathit{xl}.s}\rangle$
$\quad s\qquad\overline{\rho'[\mathit{xl} \leftarrow \mathit{tl}]}\qquad\qquad\overline{t}\qquad\qquad\cdot$